DDT Training Session

Reading Materials and Exercises

January 2011

DDT Training Course

Objectives

This training course contains the materials you will need to get started with Allinea DDT.

If you are in a taught DDT tutorial, the tutor will begin by showing you many of the DDT features as a quick overview – the exercises cover a lot of DDT, but it's nice to see a little tour up front of what to expect as the day progresses.

There are a series of walked through examples – each followed by a hands on exercise for you to try.

All of these materials will be left behind for you to use after the day is complete, so you can return to them any time. Please feel free to mail your tutor, or support@allinea.com afterwards if there is anything that you still want to know!

Session 0 – Getting Started

DDT will have been configured to work with your system already. Your tutor today will tell you what to do to get DDT started, if you do not already know.

The examples in this course all have a makefile, which should work on most systems.

If your command for MPI source compilation is “mpicc” or “mpif90” then to make an example, use “make” alone.

```
make
```

NOTE: if your MPI compilation command is different you will need to set the CC or F90 variables – the following will work on a Cray XT or XE system for example.

```
make CC=cc
make F90=ftn
```

Some platforms may require additional flags to enable memory debugging during compilation – notably AIX, BlueGene/P and the Cray XT systems use static linking. There is information in the userguide about how to link in memory debugging support on those systems – other systems do not require you to do anything.

Session 1: Straightforward Crashes

First let's look at debugging crashes – the kinds of crashes or errors that happen repeatedly. These are often segmentation faults or aborts, but even exiting with an error code.

This form of bug is very common – and very easy to fix with a debugger, but much harder without one!

We'll all walk through one case using the cstartmpi example together. This is a messy, confusing C program, with some bugs.

Afterwards, there's another crash for you to solve on your own or with your neighbour.

Walkthrough

First we will compile the application cstartmpi. There is a makefile for this in the cstartmpi directory.

```
cd cstartmpi
make
```

Run with 4 processes - it's ok

```
mpirun -np 4 ./cstartmpi.exe
```

Now try again with some arguments

```
mpirun -np 4 ./cstartmpi.exe some input arguments
```

The program will abort as there has been a problem:

```
rank 0 in job 52 tenku_60773 caused collective abort of all ranks
```

The next step is to bring this up in DDT and find out what happened. The quickest way to start is to run DDT almost identically to the way you launched MPI.

```
ddt -start  -np 4 ./cstartmpi.exe some input arguments
```

The DDT GUI will appear - and it will have started your program. You can see the source code, and there is a colour highlighted line. This is the current location that processes are at. Initially all processes are paused after MPI_Init.

At the top of DDT you will see a number of control buttons, a bit like a VCR (or PVR for the modern reader). If you hover the mouse over the control buttons, a tooltip will appear that gives the name of the button.

- Play – make the processes in the current group run until they are stopped.
- Pause – cause the processes in the current group to pause, allowing you to examine them.
- Add Breakpoint – adds a breakpoint at a line of code, or a function, that will cause processes to pause as soon as they reach that location.
- Step Into – will either step the current process group by a single line, or if the line is involves a function call, it will step into the function instead.
- Step Over – will step the current process group by a single line.

- Step Out – will run the current process group to the end of their current function, and return to the calling location.

Press play to run the program. *DDT stops with an error message – indicating a segmentation fault.*

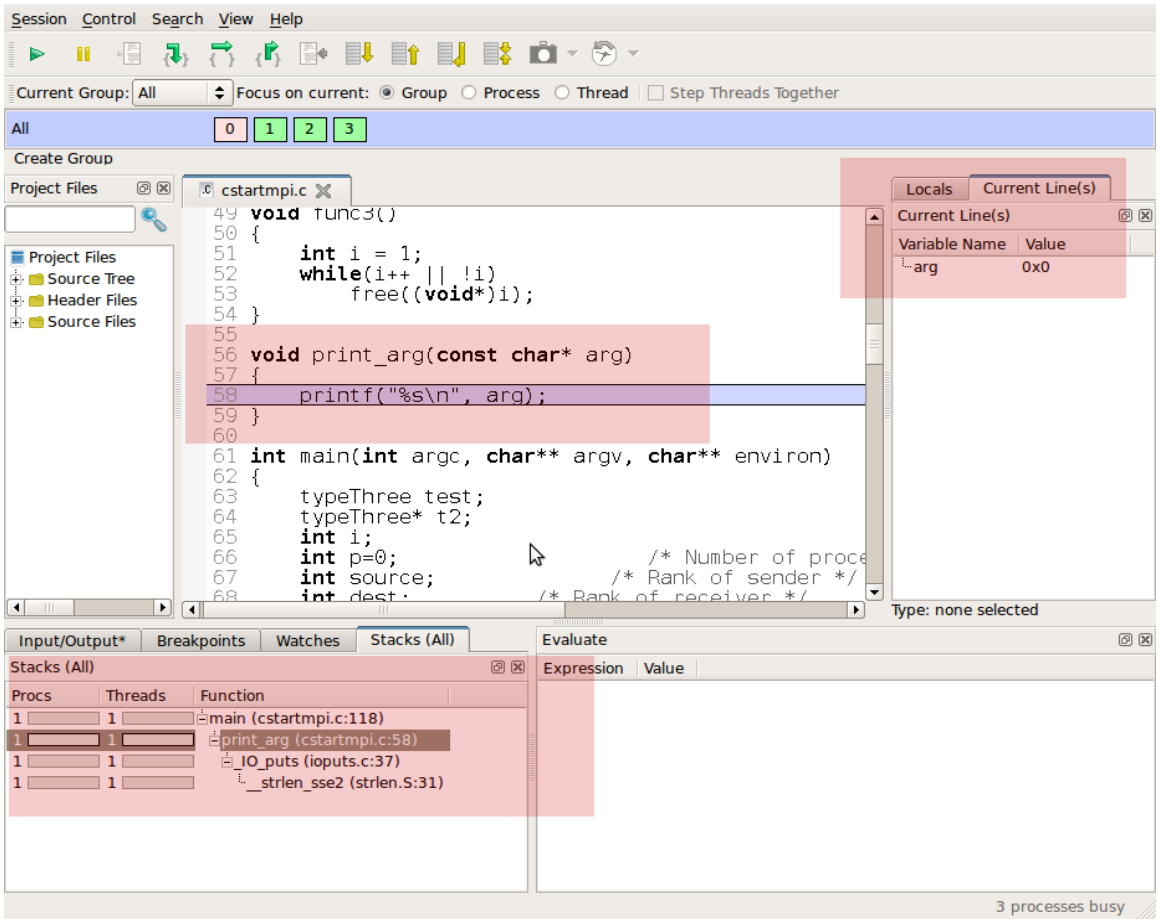

Illustration 1: DDT display after dismissing the dialog

The screenshot shows DDT after the dialog has been dismissed – we've colour highlighted the most important parts.

At the bottom of the GUI you can see the Stacks view (you may need to raise the tab by clicking on it to see it). This is tightly connected to the source code, also highlights in the screenshot. and shows where all the paused processes are: all the current function calls – higher points of the tree call the lower branches.

Often just looking at the variables at different points in the stack is enough to tell you why the program crashed.

In this case – you can see arg is the null pointer (0x0) which is invalid for its usage in the printf statement in the code. Hence, the program crashed because print_arg was called with the wrong thing.

Click on the "main" directly above the print_arg function in the Stack View. *This takes you to main which lets you see where that arg value comes*

from.

Now click on the "Locals" tab (on the right-hand side of the GUI) – you are seeing all the local variables.

Click on the "Current Line" tab to simplify and show only the variables on that line.

Click and drag between lines 113 and 118 in the source code to show all the variables in that region.

You can now see y is clearly incorrect - there aren't that many arguments (argc).

To find why is it wrong, examine the line 117: x is being checked against argc but y is being incremented.

Fix the for loop condition in your favourite editor to read "y < argc" then recompile and re-run - now it works!

Exercise

Our cstartmpi program has another bug: it runs fine for 4 processes, as we've just seen, but at larger numbers it segfaults again.

```
mpirun -np 5 ./cstartmpi.exe
```

```
rank 4 in job 60  tenku_60773   caused collective abort of
all ranks
```

Now it's up to you to find out why – you can join with your neighbour at one computer to run the program with DDT and work out what's going wrong and whether you can fix it!

Hints

- To start debugging with DDT

```
ddt -start -np 5 ./cstartmpi
```

- Click 'Play' to run a program
- Use the stack view to see which sequence functions called each other in
- Click and drag to show variables from many lines in the current line view
- Why didn't the loop terminate? Why did it terminate for the other processes?

Session 2: Deadlock in MPI

Early versions of MPI programs often get into deadlock – things such as each process waiting for another, communicators are not matched up or tags not matched. In some cases, livelock happens too – where processes are communicating but not proceeding in any useful way.

DDT can inspect the message queues to show which processes are waiting and why, in addition, simply pausing processes is a good way of finding where processes are.

We'll start with a step-by-step walkthrough and then move on to an exercise again!

Walkthrough

First we will build the cpi example.

```
cd cpi
make
```

Run it with 4 processes

```
mpirun -np 4 ./cpi
```

It works fine

```
. . . .
pi is approximately 3.1416009869231249, Error is
0.0000083333333318
wall clock time = 1.268749
```

Now we run with 10 processes and it also works fine.

```
mpirun -np 10 ./cpi
```

The next test is to try 8 processes.

```
mpirun -np 8 ./cpi
```

It locks up!

```
Process 7 on localhost
Process 5 on localhost
Process 6 on localhost
Process 3 on localhost
Process 2 on localhost
Process 0 on localhost
Process 1 on localhost
Process 4 on localhost
```

Press ctrl-c to abort, and let's try it under DDT.

```
ddt -start -np 8 ./cpi
```

When DDT returns with your code begin running the program

Press the play button.

After a while, the program has still not terminated – but the debugger has still not helped yet.

Press the pause button.

Examine the source code view and the stacks view. Both are showign that half of the processes are at one location and half at another. Half are in an MPI_Barrier and the other half are in MPI_Bcast.

The next challenge is to find out why this has happened.

Let's look at the loop used by the barrier processes. Did all processes execute it the same number of times?

> *Open the View Menu, and select the Cross Process Comparison tool.*
> *Ask DDT to evaluate "i <= n " in this dialog.*

Sure enough, the "barrier" processes are still trying to loop and the rest have already exited.

How many times should each process execute this loop?

> *Use the Cross Process Comparison to evaluate (n - (myid + 1))/numprocs*

We see that processes 0-3 execute the loop one extra time. Possible solutions are to move the Barrier out of the loop to a place where it's executed the same number of times by every process, or to modify the loop to make sure all processes execute it the same number of times

Exercise

Let's look at a new program that also deadlocks. Compile and run the Loop example.

```
cd Loop
make
mpiexec -np 8 ./loop
```

It's supposed to pass a message around the loop, but it never finishes!

Kill it and debug it with DDT – try to find what the problem is.

Hints

- A small job is ample to find the cause

```
ddt -start -np 8 ./loop
```

- Investigate the odd process out; what should it have done?
- Think of the example as passing a token around a loop 'max' times. Where does the token start? Where does it stop? What should happen to it at the end?
- Look at the "received" variable in Cross Process Comparison tool, which is the number of times the token has been received.

For an example of MPI ambiguity, replace the BUFSIZE definition with a smaller quantity (~100 instead of 1024x1024) – on most MPIs an MPI_Send of small volumes of data is completed asynchronously! This means the code would terminate successfully, even though we know there is a bug, for smaller message payloads.

Session 3: Incorrect Results

We're going to take a naive C matrix multiplication example and use it to look at some of DDT's graphing features.

This also tours some of the most important parts of debugging - breakpoints, watches, evaluation and array viewing - classic tools to really dig around in the code.

Walkthrough

We will work with the matrix example – it is a simple C code to do matrix multiplication. **Note that in this example at this point, the bug is not important, we are only walking through the feature.** Don't worry, there is an exercise with a real bug in a minute!

```
cd Matrix
make
ddt -start ./matrix
```

DDT will look slightly different from the previous runs, as it is now debugging a scalar (non-MPI application).

Hunting down the source of invalid results is a much freer activity than trac king down a crash. You often want to explore bits of code in more detail. DDT includes several features to make this easier. One is 'run to here' - a short-cut to run the program until all processes reach a certain point.

Right-click on line 29 and choose run to here.

You'll notice we can explore arrays in the local current line, but is not the best way to look at larger arrays.

Switch to the current line view, drag A in the source code into view.
Now expand A.

You can also keep an eye on an actual element within the array by using the Evaluate window.

Drag and drop B[0][2] from the Current Line tab to the Evaluate tab (bottom right of the DDT screen by default).

You can edit values as well with DDT.

Right click on B[0][2] in the Evaluate tab and select Edit Value.
Give B[0][2] a new value.

The Evaluate tab lets you also enter arbitrary expressions – like B[0][2] + A[1][1] – in addition to simple values.

Step into init_array(B, 2).
Step over a few times.

Observe how the value of B[0][2] is updated.

There are however better ways to look at a whole array.

Right-click on B in the source code (line 20).
Select View Array (MDA).

Enter B[$i][$j] as the expression and give i and j some bounds (they are inclusive bounds).

DDT also lets you visualize arrays.

Click on Visualize in 3D

The 3D view shows the array in current state of being assigned to.

Step out to finish the function init_array.
Click Evaluate in the "MDA" again and then Visualize

The array B is now being shown as it stands after it has been initialized.

Use step over to show stepping to the next line without going into any functions

Exercise

Exit DDT and run the program from the command line.

As you can see, the program runs – but it runs incorrectly. Now it's up to you to find out why!

There are many ways to find the bug. The simplest is to start DDT and add a watch for C[1][1]. This shows that it changes to 1 in the init function and gives a hint that this is a bad thing (0 is expected).

Alternatively, just looking at the C array any time before or during calculation should suggest what's happened.

Session 4: Incorrect Results

The next example is for Fortran users – it has two bugs, both are left as an exercise for you.

The code we will use is the Array example.

```
cd Array
make
```

Exercise

The code is a simple convolution code. It is an MPI code, although it doesn't do any communication – and one or two processes is enough to show the problem.

A matrix B has a 3x3 so-called convolution matrix M applied to each cell. By this we mean that the new value in matrix C at cell (i,j) is the arithmetic sum of the products of each cell surrounding B(i,j), and B(i,j) itself – with a multiplication mask given by M.

Thus the 3x3 convolution matrix with all values zero except M(2,2), the central element, which is 1, is an identity matrix for convolution.

The developer of this code was kind enough to create and include a test case involving the identity convolution matrix, which applies it to an 5x5 array B – but something strange happens, the output is not the same as the input.

```
mpirun -np 8 ./array
```

The output snippets below should match values.

```
 A real convoluted example code
 Start of input b
   1.0000000       0.0000000       0.0000000       0.0000000
0.0000000
…
```

```
 Output of the convolution of b
   0.0000000       0.0000000       0.0000000       0.0000000
0.0000000
…
```

However, they are different. Can you find out why, and fix the problem?

Hint

- A good place to start is the function called convolute – and take a look at where 'c' is calculated for element (1,1).
- Once you have found where things happen, double click on the line at the start of the relevant loop to set a breakpoint there (line 161 looks a good one for this).
- Restart the example and step through the offending loop to see what happens.

You will probably kick yourself when you see the answer, maybe a different font would help – try using the Session/Options/Appearance menu and change the code viewer font settings.

- Be really careful when you fix the problem so that you change the code in both loops –

when you assign the elements of C to B, remember which elements you did not compute!

Summary

We have seen a number of features in DDT that can help you to fix the really common types of problem that occur in everyday development.

This didn't cover everything that DDT can do for you, but it should give you the confidence to use DDT and try other features as you become more familiar with it.

For example, DDT's process groups are a great way of controlling subsets of processes – they're quick and easy to create and let you set breakpoints or step, say, with only a partial set of processes.

Another example is using the attaching feature to attach to a job that is already running.

The userguide gives a more comprehensive look at the features of DDT and you can get this to appear in DDT by pressing F1, a PDF version is also available in the doc subdirectory of DDT's installation.

If there are any questions you have, or problems with using DDT, please remember that support@allinea.com exists to ensure your debugging is successful! We always like to hear from you, as only users like you can help us to know what is important in our debugger.